Hashing Lemmas on Time Complexities with Applications to Formula Manipulation

EIICHI GOTO *,** AND YASUMASA KANADA *

* DEPARTMENT OF INFORMATION SCIENCE, UNIVERSITY OF TOKYO, TOKYO, 113, JAPAN
** INSTITUTE FOR PHYSICAL AND CHEMICAL RESEARCH, WAKOSHI, SAITAMA, 351, JAPAN

I. Introduction and Summary

Johnson[1] and Horowitz[2] applied sorting to improve time complexity of multiplication of univariate polynomials. Their results may be regarded as applications of the following LEMMA:

Sorting LEMMA. The time complexity of sorting of N items is $O(N\log_2 N)$ and that of binary search of sorted N items is $O(\log_2 N)$.

In this paper, time complexities of operation on "sets" and "ordered n-tuples" based on a hashing table search technique are presented as "Hashing LEMMAs" and are applied to formula manipulation. Unique normal forms for multivariate symbolic formulas resulting in $O(1)$ time complexity for identity checks are presented. The logarithmic factor $\log_2 N$, characteristic to sorting algorithms, is shown to all disappear from time complexities of polynomial manipulations. Actual implementation of the hashing technique is outlined and actual timing data are presented in the appendix.

II. Hashing LEMMAs on Sets and n-Tuples.

(2.0) Denotations and Conventions:

In case x represents a set or an n-tuple, |x| means the number of elements.

Sets are denoted by underscored capital letter(s). Specially,

INT is the set of (all) integers;
INT0 = INT - {0}, i.e., integers except 0;
INT+ is the set of positive integers.

A BNF metaobject is denoted by embracketing a set in the underscoring notation between "<" and ">", with optional commentary un-underscored letters. This convention enables us to use both BNF and set notations. E.g., BIT = {0,1} and <Binary digIT> ::= 0|1 。 are equivalent definitions, where "。" means the end of a BNF definition.

In order to present algorithms precisely and concisely, Lisp with three additional data types <ordered n-TUPle>, <SET> and <ASSociator> are used in this paper. <INTeger>, <SYMbol, i.e., nonnumeric atoms> and <CONS, i.e., data created by Lisp functions "cons" or "list"> are the three data types of ordinary Lisps. (Floating point numbers and arrays are omitted because of irrelevance to this paper.) Since the time complexity of high precision arithmetic is not the theme of this paper, the time complexities of arithmetic operations on <INT>'s are assumed to be $O(1)$ for the sake of simplicity.

<IDentifiables> are defined as:
ID = INT ∪ SYM ∪ TUP ∪ SET ∪ ASS; (<CONS> ∉ ID).

While <ASS>'s are denoted as <ASS>::= (.<ID>)。, <TUP>'s and <SET>'s are denoted in accordance with ordinary mathematical notations:

<TUP> ::= (<ID>,。。。)。; <SET> ::= {<ID>,。。。}。

where ",。。。" means nonzero repetition of the same metaobject. Specially the 0-tuple () and the null set {} are regarded equivalent to NIL, i.e.,

() ≡ {} ≡ NIL.

<CONS> is printed as cons[A;()] = (b̸Ab̸) with extra blanks (b̸'s) at both ends to discriminate them from a <TUP> printed as (A).

(2.1) A function "tcons" appends an <ID> to a <TUP>, e.g.,

tcons[A;()]=(A), tcons[{A,B};(C)]=({A,B},C).

Lisp functions "car", "cdr", "cadr" etc. work on <TUP>'s as on <Lisp LIST>'s, e.g.

car[(A,B)]=A, cdr[(A,B)]=(B), cadr[(A,B)]=B.

<TUP>'s are uniquely represented in the machine by making use of hashing for speed:

LEMMA 1. The time complexities of functions "tcons", "car" and "cdr" on <TUP> are all O(1)

(2.2) A function "settup" transforms a <TUP> into a <SET> with the corresponding elements; "tupset" does the converse, e.g.,

settup[(A,B)]={A,B} or {B,A};
settup[(A,B,B)]={A,B} or {B,A};
tupset[{A,B}]=(A,B) or (B,A).

Specially for t ∈ TUP, tupset[t]=t (a coercion rule). Although the ordering of elements of a <SET> is irrelevant to its identity, the ordering of the elements of the <TUP> used first to define a <SET> establishes a "canonical order" among the elements of the <SET>. Whenever the canonical order is needed, it can be retrieved by performing tupset [<SET>]. <SET>'s are represented uniquely in the machine by making use of hashing for speed:

LEMMA 2. For t ∈ TUP, s ∈ (SET ∪ TUP), the time complexities of settup[t] and tupset[s] are O(|t|) and O(1), respectively.

(2.3) For x ∈ ID the function "ass" yields an <ASS> : ass[x]=(.x*). (* means actual datum represented by the variable). Conversely, for a=ass[x] ∈ ASS the function "key" gives the <ID>, x: key[a]= x and the pseudo-function assign[a;v] assigns a value v, of any type, to <ASS>, a. The value is assign[a;v]= v and the assigned value can be retrieved as the value of the function value[a]=v. The initial value of an <ASS> is (). Similarly to Lisp, property functions are defined as put[x;y;v]= assign[ass[tup [x;y]];v], get[x;y]=value[ass[tup[x;y]]] and remprop[x;y]=put[x;y;()], where x, y ∈ ID and v is a datum of any type. These functions are implemented by making use of hashing for speed:

LEMMA 3. The time complexities of "ass", "key", "assign", "value", "put", "get" and "remprop" are all O(1).

Note in ordinary Lisps that properties are more restrictive: x ∈ SYM and y ∈ (INT ∪ SYM), and that in case *m* properties are used on a SYM the time complexity may increase as *O(m)* due to list implementation of properties.

(2.4) For x, y ∈ ID, the predicate function eq[x;y] checks the equality of x, y in accordance with the mathematical common sense. Namely, in case x and y are of different types, eq[x;y]=(); for x, y ∈ INT, eq[x;y]=T iff x and y are numerically equal; for x, y ∈ SYM, eq[x;y]=T iff x and y have the same spelling; for x, y ∈ ASS, eq[x;y]=T iff key[x]=key[y]; for x, y ∈ TUP ∪ SET, eq[x;y]=T iff x and y represent the same n-<TUPle> or <SET> mathematically. E.g.,

eq[(A,B);(B,A)]=(), eq[{A,B};{B,A}]=T,
eq[{A,B};{B,B,A}]=T, eq[(.(A));(.{A})]=().

LEMMA 4. The time complexity of "eq" is O(1).

Note that for the equality checking of Lisp data <CONS>, the time consuming function "equal" has to be used[3]. <TUPle>'s essentially differ from <LIST>'s in this regard.

(2.5) Outline of an Implementation called HLISP (Hashed LISP).

Each <HLISP CELL> in the FSA (Free Storage Area) consists of three fields: <CELL> ::= [<TAG>,<CAR field>,<CDR field>]$_\circ$. Besides for GBC (GarBage Collection) marking, the <TAG> is used to specify the data type of the cell. Similarly to Lisp 1.5, a <CONS CELL> ::= [CONS,x*,y*]$_\circ$ is created in the FSA as the result of cons[x;y]. The FSA itself is used as the (only one) hash table with the size being a prime p. For tup[x;y], a hash search (insert iff absent) is made for a <TUP CELL> ::= [TUP,x*,y*]$_\circ$, using Knuth's algorithm D[4, p521], thereby ensuring uniqueness of the resultant <TUP>. For ass[x], a hash search is made for an <ASS CELL> ::= [ASS,"don't care",x*]$_\circ$, using Knuth's algorithm U2[4, p539]. The value of the <ASS> is placed in the <CAR field>, which is not used as the key of the hash search. A <Short INTeger> is represented as a pointer (placed in <CAR> or <CDR> field) to a non existing memory address. An n-precision <INT> is uniquely represented like a <TUP> of <Short INT>'s (i_1, i_2, ..., *in*) with the head cell being changed to an <INT CELL> ::= [INT,i_1,t]$_\circ$, where t is a <TUPle>, (i_2, ..., *in*). A <SYM CELL>, corresponding to an atom header cell of Lisp 1.5, is the same as an <INT CELL>, except the head cell <SYM CELL> ::= [SYM,i_1,t] with <Short INT>'s i_1, ..., *in* being an unique encoding of the character string which identifies the <SYM>. For settup[t], t=(e_1*, ..., *em**), a <SYS1 CELL> ::= [SYS1, "don't care", "don't care"]$_\circ$ is made first, where SYS1 is a system data tag. Secondly, a <TUP> t'=(e_1'*, ..., *en*'*), free of duplicating elements is made from t by using hash searches for <SYS2 CELLs> ::= [SYS2, "pointer to the SYS1 cell", *ei**]$_\circ$ for removing duplications with time complexity *O(1)* per element of t. Thirdly, using a symmetric (in respect to permutation of arguments) hash sequence *hi*(e_1'*, ..., *en*'*) *i*=1, 2, 3, ... (e.g., h_1=mod(e_1'*+ ... + *en*'*,p-1)+1, *hi*= mod(*i**h_1,p) with time complexity *O(n+i)*; Algorithm U2[*ibid*], [5]), hash search is made for a cell s=[SET,h_1, "don't care"]. If unsuccessful, a new <SET CELL>, <SET CELL> ::= [SET,h_1,[SYS1,|s|,t']]$_\circ$, is created. If successful, s = settup[t] (redefined <SET>) or ≠ (hash conflicting <SET>'s) is checked by utilizing the <SYS2 CELL>'s of t. (Time complexity *O(|t'|)* at the most.) The hash search is resumed in the latter case. The load factor α of the FSA is limit to $\alpha \le \alpha M < 1$ (e.g., αM=80%). When $\alpha \ge \alpha M$ the GBC is called. A trioccupancy ("occupied" (i.e., a cell in use), "deleted" (not in use but in hash conflict) and "empty" (neither in use nor in conflict)) scheme is used to reclaim the garbage <CELL>'s without cell relocations and without using secondary storage. (A detailed analysis is given in [6]; McCarthy [7], proposed a scheme essentially the same as the present uniquely represented n-<TUPles>. However, he stated a difficulty in GBC: the neccesity of the use of secondary storage.) If the result of GBC does not satisfy $\alpha < \alpha m$ (e.g., αm=60%), GGBC (Grand GBC; more details are given in IV) is called. If $\alpha < \alpha m$ is still not satisfied the job is terminated because of insufficient storage. Note that the condition $\alpha m < \alpha M < 1$ ensures the time complexities as claimed in LEMMAs 1-4. If $\alpha m = \alpha M = 1$ were used, the FSA would be usable up to the very last one cell, but the LEMMAs would not be valid.

III. Application to Formula Manipulation.

Let IP be the set of polynomials with integer coefficients and positive integer exponents.

(3.1) The <Sum of Product> Normal Form.

Polynomials of IP can be expressed as sum of products (terms), e.g.,

$$p1 = 2UV^2 + 3X^3Y^4,\ p2 = 3Y^4X^3 + VUV + UV^2.$$

These expressions represent the same polynomial, and they can be faithfully represented in terms of <TUP>'s as follows:

<SP*form>::=((<TERM ID*>,<COEFficient>),...)$_\circ$ and
<TERM ID*>::=((<VARiable ID>,<EXPonent>),...)$_\circ$,

where <COEF> ∈ INTO, <VAR ID> ∈ SYM and <EXP> ∈ INT+. E.g.,

```
sp*(p1)=((((V,2),(U,1)),2),(((X,3),(Y,4)),3))
sp*(p2)=((((Y,4),(X,3)),3),(((V,1),(U,1),(V,1)),1),
        (((U,1),(V,2)),1)).
```

These SP* forms can be transformed into a unique SP normal form in the following way (a program is given later): (1) Combine duplicating <VAR ID>'s in a <TERM ID*> as in $VUV=V^2U$. (2) Absorb the commutative nature of multiplications into a SET: <TERM ID> ::= {(<VAR ID>,<EXP>),...}$_\circ$. E.g., $V^2U=UV^2$ is absorbed as {(V,2),(U,1)}={(U,1),(V,2)}. (3) Combine duplicating <TERM ID>'s as in $V^2U+V^2U=2V^2U$. (4) Absorb the commutative nature of additions into a SET: <SP> ::= {(<TERM ID>,<COEF>), ...}$_\circ$. E.g., sp[p1]=sp[p2]={({(V,2),(U,1)},2),({(X,3),(Y,4)},3)}.

We now define two data structures, in order to formalize the definition of the <SP> form: A <CLUB> is a <SET> of 2-<TUPle>'s of <ID>'s (informally, <CLUB> ::= {..., (*mi*, *gi*), ...}$_\circ$) such that all of the first element, to be called the (club-) "member", of the 2-<TUPle>'s are distinct (*mi*≠*mj* for *i*≠*j*). The second elements (*gi*'s) of the 2-<TUPle>s are called "grade"s. A <MULTISET> is a special <CLUB> of which the grades are restricted to positive integers. (This agrees with the "multiset" of Knuth[4] by regarding the "multiplicity" as the grade.) Thus, we can now state: "An <SP> is a <CLUB> of <TERM ID>'s with non-zero integer grades, called <COEF>; a <TERM ID> is a <MULTISET> of <SYM>'s, called <VAR ID>'s; specially, for the null and constant polynomials,

sp(0)={}, sp(n)={({},n)}, where n ∈ INTO."

Since the SP form obviously represents IP polynomials uniquely, i.e., for p, q ϵ IP,

sp(p) = sp(q) (set equality) iff $p-q \equiv 0$,

by LEMMA 4 we obtain:

PROPOSITION 1. Given two IP polynomials in the SP form, the time complexity for identity checking of the two is O(1).

(3.2) Polynomial Manipulation in The SP Form:

A Property Adding Auxiliary Function:

```
addprop[g;x;v;r] = prog[[y];y:=get[g;x];
   [null[y] → prog2[put[g;x;v];r:=tcons[x;r]];
    T → put[g;x;v+y]];return[r] ].
```

Given g, x ϵ ID, v ϵ INT and r ϵ TUP, if the G-property (i.e., the value of get[g;x]) is (), "addprop" puts v on the property and appends x to r in the result, otherwise, v is added into the property. By LEMMAs 1 and 3, the time complexity is *O(1)*. Similarly, we define:

```
subprop[g;x;v;r]=addprop[g;x;-v;r].
```

A Property into Club-Grade Function:

```
clubprop0[g;r]=prog[[c;y;w];w:=r;
 A [null[w] → return[settup[c]]];y:=get[g;car[w]];
   [y≠0 → c:=tcons[tcons[car[w];tcons[y;()]];c]] ;
   remprop[g;car[w]];w:=cdr[w];go[A] ].
```

Given g ϵ ID and r, a <TUPle> of distinct <IDs>, "clubprop0" yields a club of the <IDs> with making the respective G-properties into grades and excluding 0-grade members. By LEMMAs 1, 2 and 3 and since loop A is executed |r| times, the time complexity is *O(|r| + 1)*. *1* is added to account the time *O(1)* needed in case |r| = 0, i.e., r = ().

A Club Union and Grade-Adding Function:

```
addclub[p;q]=prog[[g;r;w];g:=gensym[];w:=tupset[p];
 A [null[w] → prog2[w:=tupset[q];go[B]]];
  r:=addprop[g;caar[w];cadar[w];r];w:=cdr[w];go[A];
 B [null[w] → return[clubprop0[g;r]]];
  r:=addprop[g;caar[w];cadar[w];r];w:=cdr[w];go[B]].
```

Given clubs p, q with numerical grades, "addclub" yields a club of the union of members of p and q with the grades of common members being added in and 0-grade members being excluded from the result. A "gensym" (i.e., a unique <SYM> generated by the system) is used to avoid possible confusions of properties in the auxiliary functions. Similarly, subclub[p;q] is defined by replacing the "addprop" in the last line only by "subprop". Since loop A is repeated |p| times and loop B, |q| times and by LEMMAs 1, 2 and 3, the time complexity is *O(|p|+|q| +1)*. In case p, q ϵ SP "addclub" adds the two and gives the result in the SP normal form. Hence,

PROPOSITION 2. The time complexity of adding two polynomials p and q in the SP form is O(|p|+|q|+1). (Multivariateness has no effect.)

A Polynomial Multiplier Function:

```
mulsp[p;q]=prog[[g;r;u;v];g:=gensym[];u:=tupset[p];
 A [null[u] → return[clubprop0[r]]];v:=tupset[q];
 B [null[v] → prog2[u:=cdr[u];go[A]]];
  r:=addprop[g;addclub[caar[u];caar[v]];
              cadar[u]*cadar[v];r];v:=cdr[v];go[B]].
```

Given p, q ϵ SP, "mulsp" yields the product in the SP form. Note that "addclub" is used to multiply two <TERM ID>'s as in addclub[{(A,1),(B,2)}; {(B,3),(C,4)}]={(A,1),(B,5),(C,4)}. For s ϵ SP, let T(s) = |s|+ (total number of elements in <TERM ID>'s of s). The dominating term (clubprop0[r] is *O(|p|·|q|)* at the most) in the time complexity of "mulsp" is easily seen to be *O(|q|T(p)+|p|T(q))*, which arises from repeating the "addclub" on <TERM ID>s for |p|·|q| times in the nested loops A and B. Hence, we obtain:

PROPOSITION 3. The time complexity of multiplying p, q ϵ SP is O(|q|T(p)+|p|T(q)); specially in case each term is K-variate at the most, it is O(|p|·|q| (K+1)) and in the univariate case it is O(|p|·|q|). (Factors such as $log_2|p|$ or $log_2|q|$ are absent. Sparseness of the result has no effect.)

An SP* into SP Transformation Function:

```
intosp[p]=mulsp[p;{({},1)}]。, where {({},1)}=sp(1).
```

This works correctly because of the "coercion rule" in (2.2). Let T*(p)=|p|+(total number of elements in <TERM ID*>s of p ϵ SP*). We obtain:

PROPOSITION 4. The time complexity of transforming an IP polynomial p in an SP form into the SP normal form is O(T*(p)); specially in case the length of each term of p is K at the most, it is O(|p|·(K+1)). (If <TERM ID*> and SP* were sorted into a sorted normal form, the time complexity would be $O(|p|\cdot(log_2|p|)\cdot(K+1)log_2(K+1))$.)*

(3.3) The <Signed Absolute SP> form:

Let s=sp(p) be the SP form of a polynomial p ϵ IP. As a <SET>, s can be partitioned uniquely as s = s+ ∪ s-, wherein all grades of s+ are positive and those of s-, negative. Let -s- be the <SP> obtained by reversing all signs of grades of s-.

Definition. The <Absolute SP> form asp(p) of p is a <SET>: asp(p)={s+,-s-}; specially asp(0)={}.

PROPOSITION 5. For p, q ϵ IP,
asp(p) = asp(q) iff ($p \equiv q \vee p \equiv -q$).

Definition. The <SASP> normal form sasp(p) of p is a 2-<TUPle>: sasp(p)=(asp(p), sign(p)), where sign(p)=+1 in case the canonical order of the SET , asp(p) is tupset[asp(p)]=(s+,-s-), otherwise sign(p)=-1 (c.f., (2.2)); specially, sasp(0)=().

PROPOSITION 6. For p, q ϵ IP,
sasp(p) = sasp(q) iff $p \equiv q$.

(3.4) Unique Normal Forms for Rationals:

Let Q be the set of (all) rational numbers. Hereinafter, for q ϵ Q, we use the following obviously unique representation; if q ϵ INT ⊂ Q use the integer q itself; otherwise use the 2-<TUPle>, (a*,b*) such that a, b ϵ INT, b≥2, q=a/b and a, b are relative primes.

SP, ASP and SASP forms can be easily generalized to <QP, polynomials with rational coefficients and positive integer exponents> by changing the condition <COEF> ϵ INT0 for <IP>'s into <COEF> ϵ (Q - {0}).

Let QF be the set of rational functions with rational coefficients and integer exponents, i.e., QF={x/y| x ϵ QP, y ϵ (QP - {0})}. Any function r ϵ (QF - {0}) is known to be uniquely factorizable, except the arbitrariness of signs on the factors, as follows:

$$r = q\, p_1^{e_1} \cdots p_i^{e_i} \cdots p_k^{e_k},$$

wherein q ϵ (Q - {0}), e_i ϵ INT0 and p_i ϵ (IP - INT) such that p_i is not factorizable into elements of (IP - {-1,1}).

Definition. The <Factorized SASP> form fsasp(r) of r ϵ (QF - {0}) is a 2-<TUPle>:

fsasp(r) = ({..., (asp(p_i),e_i), ...}, $\pm$q), where

$\pm q = (\mathrm{sign}(p_1))^{e_1} \cdot\cdot (\mathrm{sign}(p_i))^{e_i} \cdot\cdot (\mathrm{sign}(p_k))^{e_k} \cdot q$;

specially, fsasp(0)=().

PROPOSITION 7. For x, y $\in$ QF,
fsasp(x) = fsasp(y) iff x $\equiv$ y.

PROPOSITION 8. For x, y $\in$ (QF - {0}),
car[fsasp(x)] = car[fsasp(y)] iff x/y $\in$ Q.

Proofs of PROPOSITIONs 5 to 8 have been omitted but they would be easy.

A Multiplier for x, y $\in$ (FSASP - {()}):

```
mulfsasp[x;y] = tcons[addclub[car[x];car[y]];
                tcons[mulq[cadr[x];cadr[y]];()]],
```

where "mulq" is a multiplication function of rational numbers. For a divider "divfsasp", replace "addclub" by "subclub" and "mulq" by a rational number divider "divq".

(3.5) Poisson series is a function as:

$$p = \sum_i a_i \cos(u_i) + \sum_j b_j \sin(v_j),$$

where a_i, u_i, b_j, v_j $\in$ QF.

A unique normal form POIS for this series can be obtained by absorbing the arbitrariness caused by cos(u) = cos(-u) and sin(v) = -sin(-v) into ASP forms: <POIS> ::= (<POIS COS>,<POIS SIN>)., wherein <POIS COS> and <POIS SIN> are clubs:

<POIS COS> ::= {(asp(u),sp(a)),...}. and
<POIS SIN> ::= {(asp(v),sp(sign(v)b)),...}.

with u $\in$ QF and a, b, v $\in$ (QF - {0}). It would be a matter of exercise to define Lisp functions to perform addition, subtraction and multiplication on POIS normal forms.

(3.6) The <Associator List SP> Form:

So far stress has been laid on unique normal forms and on time complexities. However, for improvements in actual speed of computation, constant factors neglected in time complexities must be taken into account. Although time complexities of cons[x;y] and tcons[x;y] are both *O(1)*, "cons" would actually work faster than "tcons" because of extra hashing overhead time needed in "tcons" to ensure uniqueness. Similarly, "value", "key" and "assign" would be faster than "ass" (c.f., (2.3)). The same would hold for the *O(n)* complexity for list[x_1; ,...;x_n] and settup[t] with |t|=n. It would be a reasonable strategy to use unique normal forms only where they are essentially needed. For example, in the manipulation (add, sub and multiply) of <IP>'s in the SP form, use of the unique normal forms for <TERM ID>'s is essential but use of a <SET> for sum of terms is not. Use of the following ALSP form would be better for the sake of speed: <ALSP> ::= (ỹ(.(g*,<TERM ID>)),... ỹ). For p $\in$ IP, alsp(p) is a <LIST> of <ASSociator>'s of 2-<TUPle>'s of a "gensym", g* and a <TERM ID>. <COEF>'s of the sp(p) are given as G-properties (i.e., get[g*;<i-th TERM ID>] = <i-th COEF>). Rewriting functions for SP forms in (3.2) into those for ALSP forms would be a matter of exercise. The similar applies to Poisson series: Use ASP forms for u's and v's and ALSP forms for a's and b's.

IV. Computing Schemes with Reclaimable Hash Tables

The choice between tabulation and recomputation is a basic problem in programming. While (hashed) tabulation provides the best time complexity of *O(1)* in many cases, extra storage space is needed to keep the tables.

In HLISP two features called tabulative and associative computing are provided, which enable users to utilize the full advantages of computing with hash tables. Moreover, in order to make a compromise between the space and time requirements automatically, a two staged garbage collection scheme, GBC and GGBC of (2.5), is employed. The <CELL>'s used for hash table entries in "tab-" and "assoc-comp" schemes are reclaimed by GGBC but not by GBC. Hence, these entries are termed "reclaimable". After having been reclaimed, the table entries are reconstructed on demand.

(4.1) "Tabcomp" is applied to member[x;s]=(x $\in$ s) for x $\in$ ID, s $\in$ SET and to n-way switching and selecting functions: tab$\alpha\beta$[x;a;e*] with $\alpha \in$ {a,d,q,g} and $\beta \in$ {q,g}. The value of a must be an n-<TUPle> of the form a=(..., (*mi**, *gi**), ...) and e* must be a constant <ID> datum. If x matches with *mi* ($\in$ ID), the resultant value is respectively cadr[(*mi**, *gi**)]=*gi**, cdr[(*mi**, *gi**)]=(*gi**) or (*mi**, *gi**) for α=a, d or q; for α=g the result is "GO TO *gi**". If no match, for β=q the resultant value is e* and for β=g the result is "GO TO e*".

(4.2) "Assoccomp" effectively avoids the recomputation of the same function for the same argument(s) by inserting the results of the previous computation in the reclaimable hash table entries. Evaluation of a function is made in the "assoccomp" mode by so specifying to the compiler or interpreter. By "assoccomp", the time complexity of recursive algorithms such as follows can be improved automatically without rewriting.

```
factorial[n]=fc[n]=[n=0 → 1;T → n*fc[n-1]],
fibonacci[n]=fb[n]=[n≤1 → n;T → fb[n-1]+fb[n-2]],
```
$_nC_m$=c[n;m]=[m=0 ∨ m=n → 1;T → c[n-1;m]+c[n-1;m-1]].

(4.3) *LEMMA 5. Time Complexities of Tab- and Assoc-comp features are as in the following table:*

	WITHOUT Tab- and Assoc-comp features.	WITH Tab- or Assoc-comp		
Function	TIME	INITIAL TIME	REPEATED TIME	EXTRA CELLS
member[x,s]	*O(|s|)*	*O(|s|)*	*O(1)*	|s|
tab$\alpha\beta$[x,a,e*]	*O(|a|)*	*O(|a|)*	*O(1)*	|a|+1
factorial[n]	*O(n)*	*O(n)*	*O(1)*	2n+3
fibonacci[n]	$O(1.618^n)$	*O(n)*	*O(1)*	2n+3
$_nC_m$=c[n,m]	$O(_nC_m)$	$O(n^2)$	*O(1)*	$3n^2/2$

The initial time means the time complexity immediately after a GGBC call. Extra cells are the number of <CELL>'s needed for reclaimable hash entries. E.g., repeated evaluation of fb[21]=10946 runs 30,000 times faster in HLISP by merely feeding a card "ASSOCCOMP ((FB))". clubmember[x;c]= tabqq [x;tupset[c];()] checks whether x is a member of the <CLUB>, c. The time complexity of *O(|s|·|t|)* in the pure Lisp algorithms[3] for s $\cup$ t and s $\cap$ t of sets s, t is greatly improved by applying "tabcomp" to "member" (even immediately after a GGBC call):

LEMMA 6. Time complexity of s $\cup$ t and s $\cap$ t for s, t $\in$ SET is O(|s|+|t|).

(4.4) Outline of an HLISP Implementation:

For "member" <SYS2 CELL>'s of (2.5) are utilized. When <SYS2 CELL>'s are reclaimed by GGBC, the <SYS1 CELL> is switched to a <SYS1* CELL> to indicate the necessity of reconstruction of the <SYS2 CELL>'s. For "tab$\alpha\beta$", initially (i.e., after GGBC) a <SYS3 CELL> ::= [SYS3,a*,e*]. is hash inserted (as a result of an unsuccessful search) and then

<SYS4 CELLs> ::= [SYS4,(mi*,gi*),[SYS3,a*,e*]]$_o$ are hash inserted by using a hash sequences determined by *mi*'s (not the <TUP> (*mi*, *gi*)) and the pointer to the <SYS3 CELL>. Hash retrieval is made by utilizing these <SYS3 CELL> and <SYS4 CELL>'s, which are all reclaimed by GGBC. In the assoccomp mode, a function fb[n], say, is evaluated as: First, make a hash search for <SYS5 CELL> ::= [SYS5, "don't care", t]$_o$, with t=tcons[n;FB], and if unsuccessful insert a <CELL>, [SYS5,1*,t], where 1* is a <SYStem SYMbol>, then compute fb[n] and replace 1* by fb[n] for future retrieval of fb[n]. Else if successful retrieve the value from the <CAR field>. Specially, in case the <CAR field> contains 1*, there must have been a vicious circle in the algorithm such as fb[n]=[n≤1 → n; T → fb[n]+ fb[n-1]]. Thus a message "CIRCULAR DEFINITION ERROR IN FB ..." is printed. GGBC reclaims <SYS5 CELL>'s except those containing 1*. Hence,

LEMMA 7. "Assoccomp" effectively checks circular definitions at runtime.

(4.5) For fc[n], fb[n], c[n,m] etc., "assoccomp" is more convenient than "tabcomp" since the range of argument(s) is generally not known in advance. Conversely, if "assoccomp" were used for member[x;s], say, a great number of wasteful hash entries for x $\notin$ s would be created. Thus, "tab- and assoc-comp" are complementary and each has its own raison d'être.

V. Concluding Remarks

The first version of HLISP without the SET feature has been in operation for two years[8], but with the TUP feature alone little advantage in formula manipulation could be found. The combination of SETs and TUPs is believed to have provided a really powerful tool for formula manipulation as indicated in III. Tab- and assoc-comp features would also be useful. Since the implementation of efficient hashing and garbage collection algorithms is a very specialized art, it would be better to separate them from the general users. Therefore, external specifications of such algorithms have been given as LEMMAs in this paper.

The following improvements are now in progress to make the schemes presented in this paper into truly useful tools for symbolic and algebraic computations:

(1) Writing of an efficient HLISP compiler[9].

(2) Implementation of a language system called "FLATS" which would enable us to absorb any existing algorithm written in Fortran, Lisp or Algol 60; and to write new algorithms with Tuples and Sets added to any of the three languages F, L or A, whichever the user may prefer (HLISP = ~~FLA~~TS).

(3) Design of hashing, GBC and runtime type check hardware to improve the ultimate speed of "FLATS".

The authors acknowledge Messrs. M. Terashima[10] and F. Motoyoshi[9] for their valuable contributions in implementing HLISP.

VI. References

[1] S.C. Johnson, *SIGSAM Bulletin*, 8, 3, p.63, '73.

[2] E. Horowitz, *J. ACM*, 22, 4, p.450, 1975.

[3] J. McCarthy, et al., *LISP 1.5 Programmer's Manual*, MIT press.

[4] D.E. Knuth, *The Art of Computer Programming, vol. 3*, Addison-Wesley, Reading, Mass., '73.

[5] M. Sassa and E. Goto, *A Hashing Method for Fast Set Operations*, submitted for publication.

[6] T. Gunji, *Tech. Rep. 76-03, ISD* (Information Science Department, the University of Tokyo), 1976.

[7] J. McCarthy, Page 151 of *Symbol Manipulation Languages and Technique*, D. Bobrow, ed., North-Holland, 1971.

[8] Y. Kanada, *Tech. Rep. 75-01, ISD*, 1975.

[9] F. Motoyoshi, *Tech. Rep. 76-05, ISD*, 1976.

[10] M. Terashima, *Tech. Rep. 75-03, ISD*, 1975.

[11] A.C. Hearn, *REDUCE2 User's Manual, 2nd. ed.*, Salt Lake City, Utah., 1973.

APPENDIX. Actual Timing Data for Polynomial and Poisson Series Manipulations.

REMARKS:
(1) The machine used is HITAC 8800/8700 at the Computer Centre of the University of Tokyo.
(2) The same HLISP interpreter system was used as the host system for REDUCE 2[11]. The free storage area was 75K cells in which 25K cells were reserved for <ID> objects.
(3) The data for polynomial multiplication were obtained to observe the dependence of time on n (number of terms in polynomials) and multiplicity, K. Observed times were normalized by $n^2(K+1)$ as PROPOSITION 3 predicate. Unit of time is in msec. '*' means 'not measured'.
(4) The FORTRAN data of univariate case were taken by a program with explicit code for hashing. The program is similar to the algorithm by Gustavson and Yun to be given at this SYMSAC '76. The hash area was selected to 5011 (a prime) and the hash probe sequence was given by Algorithm U2 of Knuth[4, p539].
(5) The programs in HLISP were written for the ALSP and ASP forms of (3.6).

Formulas \ n, t=resultant # of terms	4	8	16	32	4	8	16	32	4	8	16	32	
$(\sum_{i=1}^{n} A^{i}) * (\sum_{j=1}^{n} A^{j})$, t=2n-1	1.71	1.69	1.60	1.60	1.85	1.73	1.71	1.67	1.82	1.74	1.74	1.77	← HLISP
	4.42	2.95	3.97	5.45	3.67	3.50	4.43	7.20	4.65	4.04	5.54	9.10	← REDUCE
	.025	.024	.020	.016									← FORTRAN
$(\sum_{i=1}^{n} A^{i}) * (\sum_{j=1}^{n} A^{jn+1})$, t=n*n	1.76	1.74	1.72	1.73	1.98	1.78	1.76	1.80	1.81	1.80	1.79	1.84	← HLISP
	5.50	6.08	15.4	51.3	4.33	7.37	21.6	*	4.40	8.48	*	*	← REDUCE
	.025	.028	.020	.018									← FORTRAN
$(\sum_{i=1}^{n} A^{-2+3i}) * (\sum_{j=1}^{n} A^{-3+4j})$, t=7n-12	1.96	1.71	1.68	1.63	1.88	1.82	1.73	1.74	1.84	1.83	1.79	1.77	← HLISP
	5.35	5.85	8.20	14.3	5.42	6.53	10.6	*	5.16	7.64	12.2	*	← REDUCE
	.028	.025	.020	.016									← FORTRAN
K-variate	1-variate (A=X)				2-variate (A=XY)				4-variate (A=XYZU)				

Timing Data for Poisson Series Manipulation:	HLISP	REDUCE
(A1*COS(WT)+A3*COS(3*WT)+B1*SIN(WT)+B3*SIN(3*WT))**3	1587 msec	8077 msec